Tampereen yliopisto

Esa Niemi

SAMANAIKAISUUS PYTHONISSA

Saatavilla olevat rajapinnat ja niiden käyttö

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Kesäkuu 2019

Tampereen yliopisto

TIIVISTELMÄ

Esa Niemi: Samanaikaisuus Pythonissa
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikka, TkK
Kesäkuu 2019

Ohjelmoinnissa samanaikaisuus tarkoittaa ohjelman eri osien suorittamista samaan aikaan käyttöjärjestelmän tai ohjelmointikielen mekanismeja käyttäen. Näitä mekanismeja ovat prosessit, säikeet ja asynkroniset mekanismit. Python-ohjelmointikieli tarjoaa useita rajapintoja näiden mekanismien käyttöön.

Tämän työn tarkoitus on esitellä Pythonin CPython-toteutuksen standardikirjaston samanaikaisuuden käytön mahdollistavia rajapintoja ja niiden käyttöä. Lisäksi työssä käsiteltiin rajapintojen eroja ja hyviä ja huonoja puolia suorituskyvyn ja käyttökohteiden kannalta.

Työssä selvennettiin onnistuneesti rajapintojen taustalla olevia mekanismeja ja käyttöä, ja todettiin prosessipohjaisen samanaikaisuuden olevan paras ratkaisu laskentaintensiivisille ongelmille. Säikeisiin pohjautuvan samanaikaisuuden todettiin soveltuvan laitteisto- ja verkkopainotteisille ongelmille niiden keveyden takia. Concurrency.futures-moduulin rajapintojen todettiin olevan helppokäyttöisimmät ja laajimmat samanaikaisuuden suorituksen ja tulosten hallintaan. Asyncio-moduulin todettiin olevan hyödyllinen säikeiden ja prosessien käyttämisen välttämiseen.

Avainsanat: Python, CPython, samanaikaisuus, rinnakkaisuus, asynkronisuus

ABSTRACT

Esa Niemi: Concurrency in Python
Bachelor's thesis
Tampere University
Information and communications technology, BSc (Tech)
June 2019

In programming, concurrency is the process of execution of different parts of a program at the same time, using mechanisms like processes and threads provided by operating systems and asynchronous mechanisms provided by programming languages. Python provides several APIs for using these mechanisms.

The goal of this thesis is to introduce and clarify the APIs provided by the CPython reference implementation of the Python programming language through explaining the workings of the concurrent mechanisms behind them and by using comprehensive code examples. The thesis also seeks to study the differences of these APIs and their possible use cases.

The thesis successfully communicated the mechanisms behind the APIs and clarified their usage and purpose. Process based concurrency was deemed to be the best option for CPU heavy computing, whereas thread-based concurrency was found to be better in device and network intensive use cases. The concurrency.futures-module was found to have provided the best interface for using process and thread based concurrency in terms of the amount of code needed and the scope of provided functions, classes and methods for the management of the state and results of concurrent execution. The asyncio-module was found to be useful in writing concurrent code when trying to avoid using threads and processes.

Keywords: Python CPython, concurrency, parallel, asynchronous

ALKUSANAT

Haluan kiittää kandidaatintyöni valvojaa Maria Törhöstä kärsivällisyydestä ja avusta työn kirjoittamisen kanssa, Tampereen yliopiston kirjastoa tiedonhaun auttamisessa ja mahdollistamisessa, sekä ystäviäni ja perhettäni ymmärryksestä ja kannustuksesta työn tekemiseen.

Tampereella, 3.6.2019

Esa Niemi

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. SAMANAIKAISUUS 2
2.1 Prosessit 3
2.2 Säikeet 3
2.3 Coroutinet 3
2.4 Futuurit ja promiset 4
2.5 Samanaikaisuuden ongelmat ja ratkaisumekanismit 4
3. PYTHON-OHJELMOINTIKIELI 6
3.1 Käyttö ja toiminta 6
3.2 Syntaksi ja rakenne 7
4. SAMANAIKAISUUS PYTHONISSA 10
4.1 Säikeet 10
4.2 Prosessit 10
4.3 Coroutinet ja Taskit 13
4.4 Futuret 15
5. RAJAPINTOJEN KÄYTTÖKOHTEET JA VERTAILU 17
6. YHTEENVETO 18
LÄHTEET 19

LYHENTEET JA MERKINNÄT

API	engl. Application Programming Interface, rajapinta koodin kanssa vuorovaikuttamiseen
CPU	engl. Central Processing Unit, tietokoneen prosessori

1. JOHDANTO

Mooren lain mukaan transistorien määrä mikropiireissä kaksinkertaistuu joka toinen vuosi. Tämän takia tietokoneiden prosessorien ydinmäärä on kasvanut tasaisesti, mutta yksittäisen ytimen laskentatehon kasvu on hidastunut fyysisten rajoitusten kuten lämmön tuotannon takia. Ohjelmien prosessointi- ja reagointikykyvaatimusten edelleen kasvaessa, ohjelmointikielten on tarpeellista tarjota tapoja suorittaa koodia hajautetusti, rinnakkaisesti ja samanaikaisesti. Nämä tavat eivät kuitenkaan välttämättä ole yksinkertaisia valita tai käyttää.

Tämän työn tarkoitus on tarkastella Python-ohjelmointikielen standardikirjaston tarjoamia multiprocessing-, threading-, concurrency- ja asyncio-rajapintoja koodin samanaikaistamiseen ja samanaikaistamisen hallitsemiseen käyttöjärjestelmän ja ohjelmointikielen mekanismeja käyttäen. Työssä asyncio-moduulin käsittely on rajoitettu Pythonin dokumentaatiossa määriteltyyn korkeamman tason osuuteen, koska matalan tason toiminnallisuus on ”tarkoitettu kirjastojen ja ohjelmistokehyksien kehittäjille” [1]. Threading-moduulista rajataan käsiteltäväksi vain Threading-luokan käyttö. Rajapintojen käytön lisäksi työssä tutkitaan niiden soveltuvuutta eri käyttökohteisiin, vertailemalla rajapintojen käyttämien mekanismien ominaisuuksia ja rajapintojen toteutuksia.

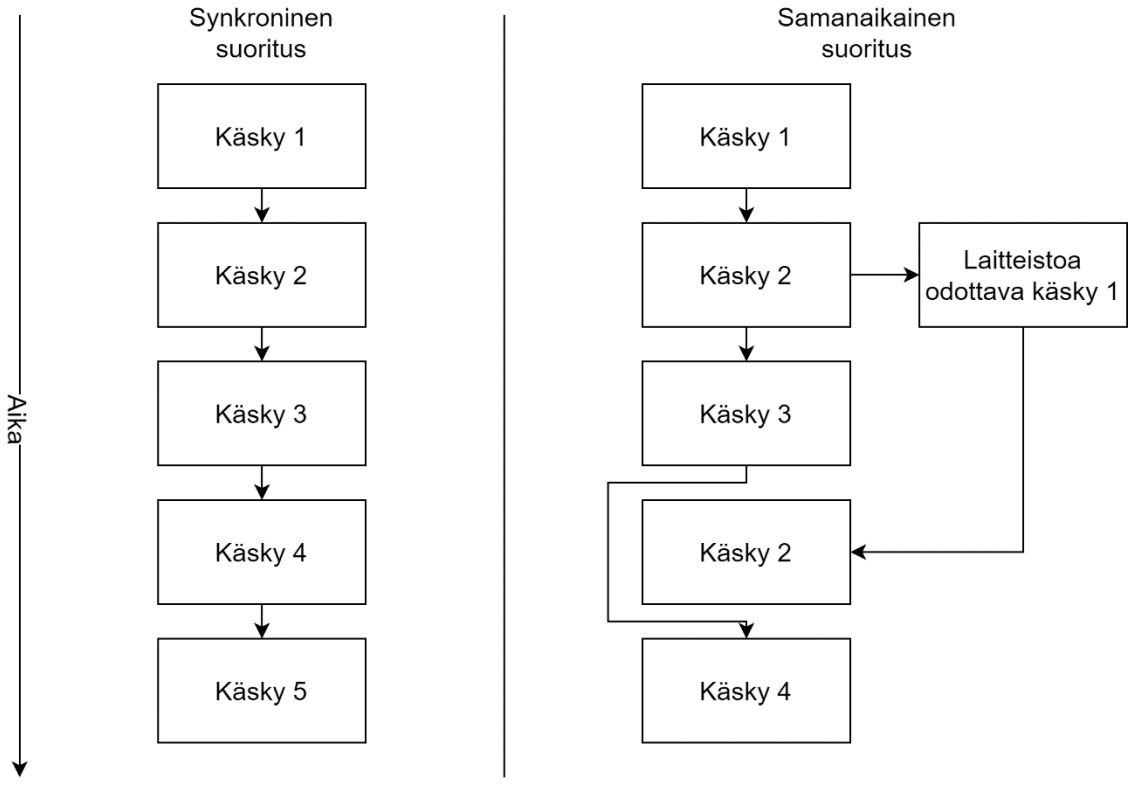
Toisessa luvussa käydään läpi samanaikaisuuden taustaa, sekä koodin suorittamisen pohjana toimivien käyttöjärjestelmän ja ohjelmointikielten tarjoamia mekanismeja samanaikaistamiseen. Lisäksi luvussa käsitellään samanaikaistamisessa ilmeneviä ongelmia ja niihin kehitettyjä ratkaisuja. Kolmannessa luvussa perehdytään Python-ohjelmointikielen toteutukseen, käyttöön, ominaisuuksiin, koodin rakenteeseen sekä syntaksiin tulevissa luvuissa käsiteltävien esimerkkien ja väitteiden ymmärtämisen helpottamiseksi. Neljännessä luvussa esitellään Pythonin samanaikaistamiseen tarkoitettuja rajapintoja, sekä koodiesimerkkejä näiden käyttöön. Viidennessä luvussa tutkitaan rajapintojen soveltuvuutta eri käyttökohteisiin toteutusyksityiskohtien sekä käytettyjen mekanismien kautta. Lopuksi kuudennessa luvussa tehdään yhteenveto työstä.

2. SAMANAIKAISUUS

Ohjelmakoodia kirjoitettaessa ohjelman käskyjen suorituksen ajatellaan yleensä tapahtuvan peräjälkeen eli synkronisesti, jolloin ohjelman suoritus siirtyy seuraavaan komentoon vasta, kun edellisen käskyn suoritus on päättynyt. Joidenkin ohjelmistokehityksen ongelmien kannalta on kuitenkin tarpeellista suorittaa useita käskyjä samanaikaisesti, esimerkiksi ajankäytön parantamiseksi tai joidenkin ohjelman osien, kuten käyttöliittymän pitämiseksi vastaanottavaisena [2].

Ohjelman eri osien, kuten Kuva 1 käskyjen suorittamista samaan aikaan voidaan nimittää useilla termeillä käytetyistä mekanismeista ja ohjelman suunnittelutavasta riippuen. Rinnakaisuus (engl. parallel computing) tarkoittaa useiden säikeiden tai prosessien käyttämistä ohjelman suorituksen jakamiseksi osiin. Rinnakkaisuus on determinististä ja tähtää yleensä parempaan suorituskykyyn. Samanaikaisuus (engl. concurrency) tarkoittaa samojen mekanismien käyttöä, mutta epädeterministisesti ja usein tavoitteena latenssin piilottaminen. Asynkronisuus taas tarkoittaa samanaikaisuuden piilottamista ohjelman kirjoittajalta. [3]

Samanaikaisuus voidaan saavuttaa useilla eri tavoilla riippuen käytössä olevasta laitteistosta sekä toteutustavasta. Näitä tapoja käsitellään seuraavissa luvuissa prosessien, säikeiden, coroutineiden, futuurien ja promiseiden osalta. Näistä coroutinet, futuurit ja promiset voidaan mieltää asynkronisiksi ominaisuuksiksi.



Kuva 1. *Esimerkki ohjelman käskyjen suorittamisesta synkronisessa ja samanaikaisessa järjestelmässä.*

Samanaikaisuudella on useita hyötyjä kuten raskaiden operaatioiden jakaminen osiin, käyttöliittymän ohjelmakoodin suorittaminen samanaikaisesti muun logiikan kanssa ja ohjelman rakenteen jakamisen helpottuminen, mutta myös haittoja kuten ohjelman suoritusjärjestyksen hahmottamisen, sekä muistinhallinnan ja virheenhaun hankaloituminen. [4]

2.1 Prosessit

Prosessit ovat käyttöjärjestelmän yksiköitä ajettaville ohjelmille, joilla on oma muistiavaruus eli alue muistista, jossa prosessi pitää suoritettavaa ohjelmakoodia ja sen tarvitsemaa tietoa. Käyttöjärjestelmä säilyttää kustakin prosessista tietoa prosessielementissä, joka sisältää tiedon prosessin ajaman ohjelman kontekstista eli ohjelman tilasta, ohjelmakoodin sijainnista muistissa, seuraavaksi ajettavan komennon muistiosoitteesta, rekistereiden sisällöstä ja tärkeimpänä tiedon itse prosessin tilasta. [2][5] Käyttöjärjestelmä käyttää prosesseja ohjelmien vuorontamiseen eli keskusyksikön suoritusajan jakamiseen vuorotellen eri ohjelmille, esimerkiksi prosessien tärkeyden, viimeisestä suoritusajan antamisesta kuluneen ajan tai prosessin tilan perusteella. [4]

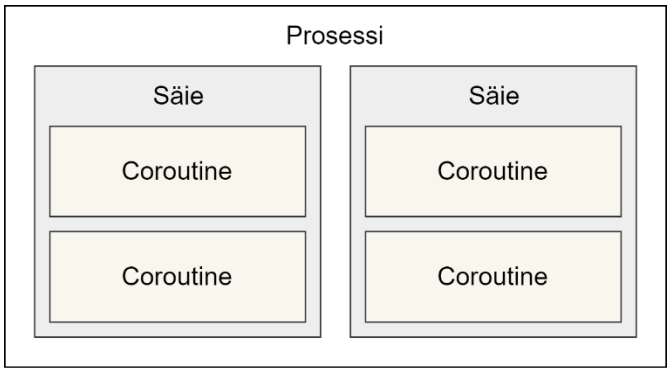
Eräitä vuoronnusalgoritmeja eli tapoja jakaa suoritusaikaa prosesseille ovat vapaaehtoinen vuoronnus ja ennakoiva vuoronnus. Vapaaehtoinen vuoronnus ei pakota prosesseja luopumaan suoritusajastaan, eli suorittamaan kontekstinvaihtoa, jossa prosessin tila ja rekisterit tallennetaan käyttöjärjestelmän muistin tilan määrän mukaan joko keskusmuistiin tai pysyväismuistiin. Sen sijaan prosessi keskeyttää ohjelman suorituksen tietyin aikavälein tai ohjelman odottaessa esimerkiksi käyttäjää tai laitteistoresursseja, jotta muita ohjelmia ja käyttöjärjestelmän tehtäviä voidaan suorittaa. [6] Ennakoiva vuoronnus taas pitää itse huolen prosessien kontekstin vaihtamisesta sille annetun vuoronnusalgoritmin perusteella, esimerkiksi prosessin tärkeyden tai arvioidun jäljellä olevan suoritusajan perusteella [7].

2.2 Säikeet

Säikeet ovat prosesseja kevyempiä ohjelman suoritusyksiköitä, joiden avulla voidaan samanaikaistaa ohjelmakoodin suorittamista käyttämällä käyttöjärjestelmän vuorontajaa. Säikeet on yleensä toteutettu prosessin osina, jotka jakavat isäntäprosessin muistiavaruuden. Näin käyttöjärjestelmän vaihtaessa suoritettavaa säiettä sen tarvitsee vaihtaa vain rekisterien sisältö, eikä koko muistiin ladattua ohjelmaa. Saman muistin jakaminen kuitenkin hankaloittaa ohjelman suunnittelua, koska säikeet saattavat muokata toisen säikeen käyttämää tietoa muistissa. Joissain käyttöjärjestelmissä ei ole erikseen säikeitä, vaan osa prosesseista jakaa muistinhallinnan. Säikeitä ja prosesseja voidaan suorittaa myös ilman käyttöliittymää tai komentolinjaa, jolloin niistä käytetään daemon-nimitystä. [4]

2.3 Coroutinet

Coroutine on aliohjelma eli ohjelman osa, kuten luokka tai funktio, joka on suunniteltu keskeyttämään suorituksensa vapaaehtoisesti ja jatkamaan sitä myöhemmin. Termi viittaa englanninkielisiin termeihin cooperative multitasking eli aikaisemmin luvussa 2.1 selitettyyn vapaaehtoiseen vuorontamiseen ja subroutine eli aliohjelmiin. Coroutinet ovat yleensä toteutettuja säikeiden osina, jolloin saman säikeen sisällä voidaan suorittaa vuorotellen useita coroutineja. Tämä mahdollistaa muun ohjelmakoodin suorituksen pitkäkestoisten operaatioiden, kuten verkko- tai laitekommunikaation suorituksen odottaessa tietoa muualta. [7] Kuva 2 havainnollistaa prosessin muistiavaruuden jakautumista eri samanaikaisuuden mekanismien kesken.



Kuva 2. *Muistiavaruuksien jakautuminen eri samanaikaistamisen mekanismien kesken.*

2.4 Futuurit ja promiset

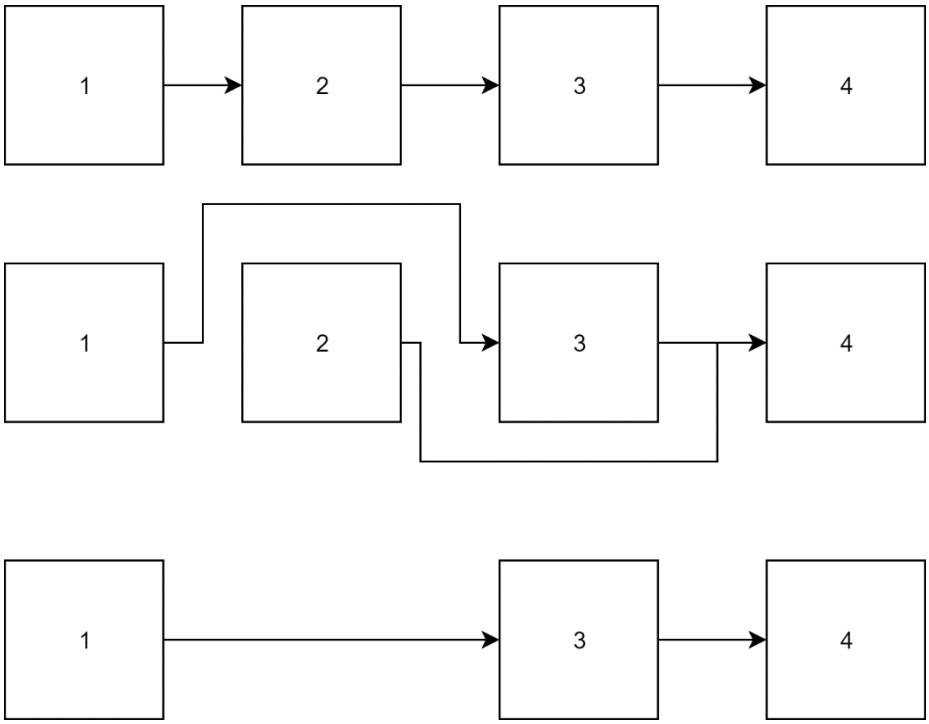
Futuurit ja promiset ovat funktionaalisesta ohjelmoinnista lähtöisin olevia ohjelmointikielen mekanismeja, joiden tehtävänä on suorittaa annettua ohjelmakoodia samanaikaisesti kutsujan kanssa ja erottaa annettu ohjelmakoodi suorituksen tuloksesta. Useimmissa ohjelmointikielissä futuurilla tarkoitetaan sijaisoliota, joka tulee sisältämään samanaikaisesti suoritettavan ohjelmakoodin tuloksen, kun ohjelmakoodin suoritus on päättynyt. Futuurien tuottaman vastauksen käsittelemiseksi futuureille voidaan yleensä määrittää takaisinkutsufunktioita, jotka suoritetaan futuurin tuloksen selvittyä. Näin minkään säikeen tai prosessin ei välttämättä tarvitse erikseen synkronoida futuureilla tai promiseilla samanaikaistettua suoritusta, vaan niiden tulokset käsitellään niiden valmistuttua. [8][9]

Promiset ovat futuurien helpompaan suorittamiseen tarkoitettu mekanismi, joka mahdollistaa useiden futuurien tuloksien käsittelyn ja yhdistämisen. Promiset tarjoavat usein samanlaisen rajapinnan kuin futuurit, jossa lopputulos annetaan promisen sisältämien futuurien suorituksen päätyttyä. [9]

2.5 Samanaikaisuuden ongelmat ja ratkaisumekanismit

Ohjelman osien samanaikainen suorittaminen ja vuorontaminen, sekä etenkin ohjelman resurssien ja muistin yhteiskäyttö aiheuttavat useita ongelmia, kuten poissulkeutuminen, synkronointi ja lukkiutuminen. Ratkaisuiksi näihin ongelmiin on kehitetty erilaisia mekanismeja prosessien ja säikeiden tilojen ja kommunikaation hallintaan. [4]

Poissulkeutumisongelmassa (engl. mutual exclusion) usea samanaikaisesti suorituksessa oleva ohjelma tai ohjelman osa yrittää käyttää yhteistä resurssia, kuten muistiin kirjoitettua arvoa tai tietorakennetta. Tämä saattaa johtaa ohjelman virheelliseen tai määrittelemättömään toimintaan. Yksi esimerkki on Kuva 3 esitetty ongelma, jossa kaksi säiettä yrittää samanaikaisesti poistaa linkitetystä listasta alkiot 2 ja 3. Poisto tapahtuu asettamalla poistettavan alkion viittaus seuraavaan alkioon osoittamaan poistettavaa alkiota seuraavaan alkioon. Kuvan keskimmäisessä tilassa säikeet ovat suorittaneet poiston samaan aikaan ja kolmannessa on lopputila, jossa alkio 3 on edelleen virheellisesti olemassa, koska alkion 1 viite seuraavaan osoittaa siihen.



Kuva 3. *Kahden peräkkäisen alkion poistaminen linkitetystä listasta kahden eri säikeen toimesta.*

Eräs ratkaisu poissulkeutumisongelmaan on semaforimuuttuja eli liikennevalo, joka asetetaan odottamaan ensimmäisen prosessin tai säikeen siirtyessä ohjelman kriittiselle alueelle eli sellaiseen kohtaan koodia, jossa käsitellään yhteisiä resursseja. Muut resursseja käyttävät prosessit tai säikeet odottavat, kunnes semafori vapautuu, jolloin niistä seuraava saa käsitellä resurssia. [4]

Synkronointiongelmassa (engl. synchronization) tarkoituksena on tahdistaa eri prosessien tai säikeiden suoritus esimerkiksi siten, että tiettyjen prosessien suoritus jatkuu vasta erään tietyn prosessin päästyä suorituksen loppuun, tai että prosessit suorittavat annettua koodia tietyssä järjestyksessä. Synkronointiongelman ratkaisemiseen voidaan käyttää esimerkiksi aktiivista odottamista, barrieria tai aikaisemmin esiteltyä semaforia. Aktiivinen odottaminen (engl. busy waiting, spinlock) toimii osittain samoin tavoin kuin semafori. Siinä kriittiselle alueelle ensimmäisenä menevä prosessi asettaa lippumuuttujan, joka pysäyttää muut säikeet, kunnes ensimmäinen on poistunut kriittiseltä alueelta. Muut säikeet odottavat loopissa, tarkistaen jatkuvasti lipun tilaa. Myös barrier estää ohjelman suorituksen jatkumisen kriittiselle alueelle, mutta sen tarkoituksena on odottaa tietyn säiemäärän saapumista odottamaan kriittiselle alueelle pääsyä, jonka jälkeen kaikki odottavat säikeet voivat jatkaa suoritusta [10]. Synkronointia tulisi välttää, mikäli mahdollista, koska säikeiden ja prosessien pitkäaikainen odotuttaminen heikentää huomattavasti samanaikaistettujen ratkaisujen tuomaa hyötyä. [4]

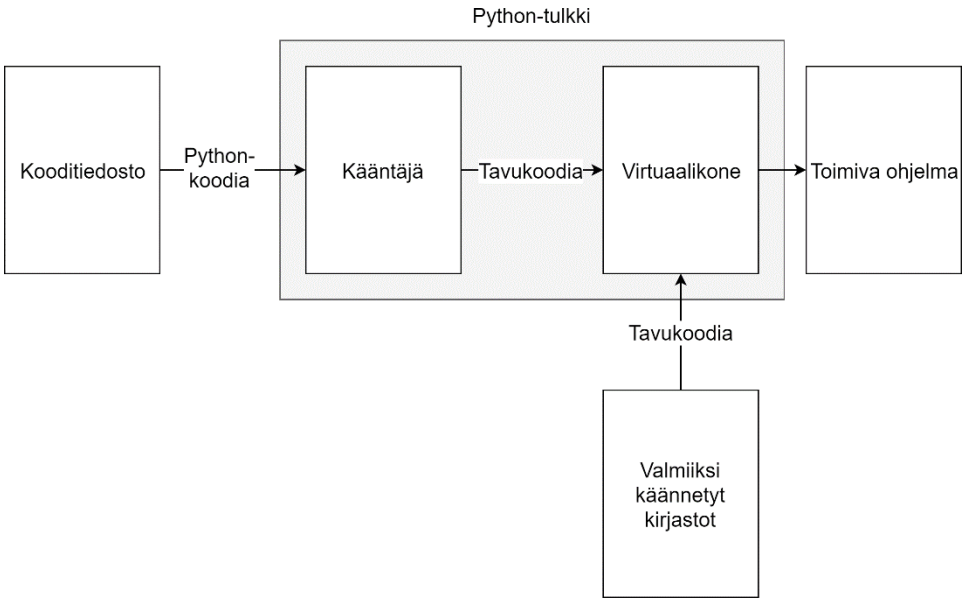
Lukkiutumisongelmassa (engl. deadlock) usea säie tai prosessi odottaa toisen odottavan prosessin lukitseman resurssin vapauttamista ikuisesti. Lukkiutuminen voidaan ratkaista purkamalla tai ennakoimalla. Lukkiutumisen purkamisessa lukkiutuminen havaitaan ja jokin säie tai prosessi pakotetaan loppumaan tai luopumaan lukitusta resurssista. Ennakoinnissa mahdollisia tapoja ovat esimerkiksi poissulkeutumisen estäminen, jossa resurssia ei voida lukita pysyvästi jollekin tietylle säikeelle tai prosessille, ja toteutus, jossa säikeet tai prosessit eivät voi lukita mitään resurssia ennen kuin ne ovat kaikki saatavilla. [4]

3. PYTHON-OHJELMOINTIKIELI

Python on ensimmäisen kerran vuonna 1991 julkaistu moderni, avoimen lähdekoodin ja korkean tason dynaamisesti tyypitetty ohjelmointikieli, jonka luojan katsotaan olevan alankomaalainen Guido van Rossum. [11] Tarkoitukseltaan Python on yleiskäyttöinen, eli sillä ei ole tiettyä suunniteltua käyttökohdetta, vaan sitä on mahdollista käyttää useisiin eri käyttökohteisiin useilla eri alustoilla. Muita sen suunnittelua ja kehitystä ajavia periaatteita ovat lukemisen ja kirjoittamisen helppous, sekä ”batteries included” -periaate, jonka mukaan Python pyrkii tarjoamaan mahdollisimman monia yleisesti tarvittavia rajapintoja valmiiksi kielen mukana tulevassa standardikirjastossa. Python on toteutettu C-kielellä ja sisältää myös rajapinnan C:llä kirjoitettujen kirjastojen käyttämiseksi Pythonilla. [12][13][14]

3.1 Käyttö ja toiminta

Vaikka Pythonia kutsutaan tulkattavaksi ohjelmointikieleksi, sillä kirjoitetun koodin suorittaminen ei vastaa täysin tulkattavaa tapaa, jossa erillinen ohjelma eli tulkki suorittaa koodia rivi riviltä. Sen sijaan Pythonilla kirjoitettu koodi esikäännetään ensin tavukoodiksi (engl. bytecode), joka tulkataan erillisen virtuaalikoneen eli varsinaisen tulkin sisällä, kuten Kuva 4:ssä. Lisäksi kirjoitettuun koodiin Pythonin import-lauseketta käyttäen lisätyt ulkoiset moduulit käännetään valmiiksi .pyc-päätteisiksi tiedostoiksi, joista ne voidaan ladata valmiiksi tavukoodimuodossa. Tämä nopeuttaa kirjastojen lataamista useilla peräkkäisillä ajokerroilla. [15][16][17]



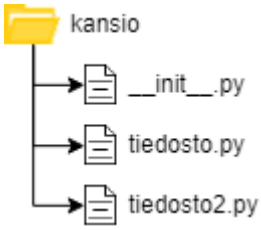
Kuva 4. *Python-tulkin toiminta. [16]*

Python-ohjelma suoritetaan antamalla tulkkiohjelmalle .py-päätteinen tiedosto, joka sisältää kielellä kirjoitetun ohjelman lähtöpisteen. CPython-tulkin asennuspaketin tapauksessa tulkki voidaan herättää komentolinjalta komennolla ”python”, jonka perään kirjoitetaan ajettavan tiedoston polku. [18]

Python on dynaamisesti tyypitetty, eli koodia kirjoittaessa muuttujille ei erikseen määritellä tyyppiä. Sen sijaan tulkki automaattisesti määrittelee kunkin muuttujan tyypin tämän senhetkisen arvon perusteella. [19] Python ei ole riippuvainen työkaluista, joilla sitä kirjoitetaan, vaan ne voivat olla yksinkertaisia tekstieditoreita kuten notepad tai xed, tai omia työkalujaan sisältäviä ohjelmointiympäristöjä, kuten JetBrains PyCharm tai CPython-asennuspaketin mukana tuleva IDLE.

3.2 Syntaksi ja rakenne

Pythonilla kirjoitettu koodi jaetaan paketeiksi ja moduuleiksi tiedostorakenteen perusteella. Kansio, joka sisältää __init__.py -nimisen tiedoston muodostaa paketin. Yksittäiset tai paketin sisällä olevat .py-tiedostopäätteiset kooditiedostot ovat moduuleita. Kuva 5 esittää kansio-nimisen paketin sisältämiä tiedosto ja tiedosto2 -moduuleja.



Kuva 5. *Esimerkki Python-paketin kansiorakenteesta.*

Moduulit koostuvat muuttujien, luokkien ja metodien määrittelyistä, metodien kutsumisesta ja pakettien ja moduulien lataamisesta. [20] Esimerkissä Ohjelma 1 on esitetty kokonaisluku, merkkijono ja totuusarvomuuttujien, metodin ja luokan määrittely. Lisäksi esimerkin alussa ladataan time-moduulista sleep-funktio, jota käytetään myöhemmin esimerkin nuku-funktiossa, ja käsitellään luokkainstanssin a jäsenmetodia tulosta_aika pistenotaation avulla rivillä 47. Alla olevassa esimerkissä funktioiden ja metodien argumentit tunnistetaan niiden järjestyksen perusteella, mutta ne on myös mahdollista nimettyinä argumentteina (engl. keyword argument). Tällöin argumentti annettaisiin muodossa ”argumentin_nimi=argumentti”. Import-lauseke hakee annettua moduulia ajoympäristöstä sekä ympäristömuuttujilla määritetyistä poluista ja lataa sen käytettäväksi ohjelmaan annetulla nimellä. Lausekkeesta voidaan käyttää myös muotoa ”from moduuli import funktio”, jolloin funktiota käyttämiseksi ei tarvitse käyttää paketin nimeä ja pisteoperaattoria. [21].

```
# -*- coding: utf-8 -*-
# määritellään tiedoston merkistö, jotta kaikki aakkoset
# tallentuvat oikein

# ladataan standardikirjaston moduulista time metodi sleep
from time import sleep


# funktion määrittely
def nuku(aika):
    # asetetaan ohjelma odottamaan annettu aika sekunteissa
    sleep(aika)


# luokan määrittely
class Aika:

    # luokkamuuttuja joka on kaikille luokan instansseille yhteinen
    etuliite = 'Aika: '

    # __init__ on luokan alustusmetodi, jota kutsutaan aina kun
    # luokasta alustetaan instanssi
    def __init__(self, aikaleima):
        # asetetaan luokan instanssimuuttuja
        # luokan sisällä luokan omaa kontekstia voidaan käsitellä
        # self. -syntaksin kautta
        self.aikaleima = aikaleima

    # luokkametodin määrittely
    def tulosta_aika(self):
        # tulostetaan luokkamuuttuja "etuliite" ja luokan
```

```
32              # instanssimuuttuja "aikaleima" komentolinjalle
                print(Aika.etuliite + self.aikaleima)
34
            def nollaa_aika(self):
36              self.aikaleima = ''

38
    # suoritetaan seuraava koodi vain
40  # jos ohjelma käynnistetään tästä
    # moduulista
42  if __name__ == '__main__':
            # kutsutaan funktiota nuku argumentilla 10
44          nuku(10)

46          # alustetaan instanssi Aika-luokasta ja talletetaan se
            # muuttujaan a
48          a = Aika('30.5.2019')

50          # luokkainstanssin metodin kutsuminen
            a.tulosta_aika()
52
    # ohjelma tulostaa:
54  # Aika: 30.5.2019
```

Ohjelma 1. *Esimerkki muuttujien, funktioiden ja luokkien määrittelystä ja käytöstä*

Tiedostossa olevaa Python-kooditiedosto käsitellään lohkoittain ylhäältä alas. Lohkot erotellaan sisentämällä, jolloin jokaista sisäkkäistä luokkaa, metodia tai funktiota kohden koodirivin alkua sisennetään neljä välilyöntiä [22]. Ensimmäisenä käsitellään moduulin tai luokan sisentämätön koodi. Esimerkkikoodissa tämä näkyy esimerkiksi Aika-luokan ja sen jäsenmetodien sisennyksissä. Alla oleva lista selventää koodiesimerkin suoritusjärjestystä. Erityisesti huomioitavaa on kohdan 5 luokkamuuttujan alustaminen, joka tapahtuu ennen luokan alustusmetodin kutsumista.

1. from time import sleep
2. nuku(10)
3. sleep(10)
4. a = Aika(’30.5.2019’)
5. etuliite = ’Aika :’
6. __init__(self, aikaleima)
7. self.aikaleima = aikaleima
8. a.tulosta_aika()
9. print(Aika.etuliite + self.aikaleima)

Jos yllä oleva esimerkkimoduuli ladattaisiin toisessa moduulissa import-lauseketta käyttäen, kääntäjä kutsuisi riveillä 38-45 olevia kutsuja ja alustuksia, muttei käsittelisi Aika-luokan nollaa_aika-metodia, koska sitä ei kutsuta millään rivillä. Metodien ja luokkien koodi käsitellään vasta, kun niitä ensimmäisen kerran käytetään. [23]

Python tarjoaa ohjelman kulun muuttamiseen toisto- ja valintarakenteita, joista tulevissa luvuissa käytetään for- ja if-lausekkeita. For mahdollistaa tietorakenteiden tai range-funktiolla määriteltyjen numerosarjojen yli iteroimisen, eli lausekkeen sisältämän koodin suorittamisen jokaiselle tietorakenteen tai sarjan alkiolle. Kullakin hetkellä käsiteltävää alkiota voidaan käyttää lausekkeen sisältämässä koodissa sen muuttujan nimellä. If on ehtolauseke, joka suorittaa sisällään olevan

koodilohkon, jos if-sanaa seuraava lauseke palauttaa toden arvon. Kahden olion vertailemiseksi toisiinsa voidaan käyttää ”==”-vertailuoperaattoria, tai pelkkää olion nimeä, jos olio palautta totuusarvon. Alla oleva esimerkki Ohjelma 2 havainnollistaa for- ja if-lausekkeiden käyttöä.

```
# -*- coding: utf-8 -*-

# luodaan range-funktiolla numeroväli
# ja tulostetaan jokainen välillä
# oleva numero
for i in range(0, 3):
    print(i)

# alustetaan vektori ja tulostetaan
# jokainen sen arvo for-lauseketta
# käyttäen
vektori = [1, 2, 3]

for i in vektori:
    print(i)

# alustetaan muuttuja jonka arvo on tosi
# ja tulostetaan merkkijono "Tosi" jos
# muuttujan arvo on tosi
arvo = True

if arvo == True:
    print('Tosi')

# ohjelma tulostaa:
# 0
# 1
# 2
# 1
# 2
# 3
# Tosi
```

Ohjelma 2. *For- ja if-lausekkeiden käyttö*

4. SAMANAIKAISUUS PYTHONISSA

Pythonin standardikirjasto tarjoaa useita rajapintoja eri tasoisen samanaikaisuuden, kuten säikeiden, prosessien ja useisiin moderneihin ohjelmointikieliin viime aikoina lisättyihin taskeihin, coroutineihin ja futureihin. [24][25][26][27] Seuraavissa luvuissa käsitellään näiden samanaikaisuuden mekanismien toteutuksia Pythonin standardikirjastossa ja esitetään esimerkkejä niiden käytöstä.

4.1 Säikeet

Pythonin säikeitä käyttävän samanaikaisuuden rajapinta on toteutettu threading-moduuliin. Ajettavaa säiettä vastaa Thread-luokka, jolle annetaan alustettaessa suoritettava funktio, kuten alla oleva Ohjelma 3, joka tulostaa merkkijonon toisessa säikeessä.

```
# -*- coding: utf-8 -*-
from threading import Thread


def tulosta(merkkijono):
    print(merkkijono+' säikeessä')


if __name__ == "__main__":
    # luodaan säieolio suorittamaan tulosta-funktio
    # "Merkkijono"-argumentilla
    saie = Thread(target=tulosta, args=('Merkkijono',))
    # käynnistetään säie
    saie.start()
    # odotetaan säikeen suorituksen päättymistä
    saie.join()

# ohjelma tulostaa:
# Merkkijono säikeessä
```

Ohjelma 3. *Yksittäisen säikeen luominen funktion suorittamiseksi*

Thread-luokalla luotu säie on mahdollista käynnistää daemon- eli taustasäikeenä, jonka suoritus päättyy pääsäikeen eli ohjelman suorituksen aloittaneen säikeen lopettaessa. Käynnistäminen tapahtuu joko Thread-oliota alustaessa avainsana-argumentilla ”daemon=True”, tai olion alustamisen jälkeen asettamalla olion daemon-jäsenmuuttujan arvo todeksi ennen käynnistämistä. Lisäksi threading-moduuli sisältää samanaikaistamista helpottavia luokkia ja funktioita. [28]

4.2 Prosessit

Pythonin standardikirjasto tarjoaa prosessien luomiseen ja käyttöön multiprocessing-moduulin, jonka rajapinta vastaa pitkälti threading-moduulia. Multiprocessing-moduulissa käyttöjärjestelmän prosessia vastaa Process-luokka, jonka avulla voidaan hallita Python-koodin suorittamisesta eri prosesseissa. Alla olevassa esimerkissä rivillä 14 alustetaan Process-luokan olio, asetetaan se suorittamaan annettua funktiota ”foo” ja käynnistetään prosessi. Foo-funktion toteutus asettaa sitä suorittavan prosessin odottamaan funktiolle argumenttina annetun viiveen, tässä esimerkissä 10 sekunnin, ajan. Käynnistetyltä prosessilta voidaan hakea sen PID-tunnus, eli käyttöjärjestelmän antama uniikki tunnus, jolla prosessia voidaan hallita käyttöjärjestelmän prosessirajapintojen ja työkalujen kautta.

```
# -*- coding: utf-8 -*-
from multiprocessing import Process
import time


def foo(viive):
    # asetetaan prosessi odottamaan annetun viiveen ajan
    time.sleep(viive)
    print('Prosessi on valmis')


def main():
    # luodaan uusi Process-olio, jonka suoritettavana funktiona on
    # foo ja argumenttina 10
    proc = Process(target=foo, args=(10,))
    # käynnistetään uusi prosessi
    # vasta tämä luo uuden prosessin
    proc.start()
    print(proc.pid)
    # odotetaan nykyisessä prosessissa, kunnes uusi prosessi on
    # valmis
    proc.join()


if __name__ == '__main__':
    # suoritetaan ohjelman pääfunktio
    main()

# ohjelma tulostaa:
# 9096
# Prosessi on valmis
```

Ohjelma 4. *Esimerkki uuden prosessin luomisesta ja käynnistämisestä Process-luokkaa käyttäen.*

Käyttöjärjestelmästä riippuen uusi prosessi voidaan käynnistää joko täysin uutena prosessina, tai nykyisen prosessin aliprosessina, jolloin kaikki isäntäprosessin tiedot kopioituvat aliprosessiin. Käynnistysmetodi voidaan asettaa kutsumalla multiprocessing-moduulin set_start_method -funktiota, joka ottaa argumenttina käynnistystavan nimen.

Multiprocessing tarjoaa myös työkaluja useiden prosessien helpompaan käyttöön ja prosessien väliseen kommunikointiin. Pool-luokkaa voidaan käyttää annetun funktion, metodin tai tietorakenteen käsittelyyn luokalle annetuilla prosesseilla. Alla olevassa esimerkissä Ohjelma 5 on toteutettu ohjelma, joka tulostaa luvun neliön luvuille 1-3 ensin yksittäisiä prosesseja ja riviltä 38 alkaen Pool-luokkaa käyttäen.

```
# -*- coding: utf-8 -*-
from multiprocessing import Pool, Process


# palautetaan luvun neliö
def nelio(numero):
    return numero * numero


# tulostetaan luvun neliö
def tulosta_nelio(numero):
    print(numero*numero)


```

```
    # suoritetaan jos tämä moduuli on ensimmäinen ladattu moduuli
16  if __name__ == '__main__':
        # alustetaan vektori luvuista jotka tullaan neliöimaan
18      numerot = [5, 10, 20]
        # alustetaan tyhjä vektori prosessiolioiden
20      # säilyttämistä varten
        prosessit = []
22
        # iteroidaan numerolistaa
24      for numero in numerot:
            # käynnistetään uusi prosessi jokaisen numeron
26          # neliön laskemiseksi
            prosessi = Process(target=tulosta_nelio, args=(numero,))
28          # lisätään prosessiolio prosessivektoriin
            prosessit.append(prosessi)
30          # käynnistetään prosessi
            prosessi.start()
32
        # iteroidaan prosessilistaa ja odotetaan vuorollaan
34      # jokaisen prosessin valmistumista
        for proc in prosessit:
36          proc.join()

38      # luodaan Pool-olio jolla on 3 prosessia
        pooli = Pool(processes=3)
40      # kutsutaan poolin map-funkiota joka kutsuu jokaiselle
        # numerot-tietorakenteen alkiolle nelio-funktiota
42      # ja palauttaa tuloksen vektorina
        # jokainen prosessi saa siis yhden funktion suoritettavakseen
44      print(pooli.map(nelio, numerot))

46  # ohjelma tulostaa:
    # 100
48  # 25
    # 400
50  # [25, 100, 400]
```

Ohjelma 5. *Useiden prosessien hallinta Pool-luokalla*

Queue-luokka ja Pipe-funktio mahdollistavat prosessien välisen kommunikaation tietorakenteita tai viestikanavia käyttäen. Queue on standardikirjaston samannimistä luokkaa vastaava toteutus, eli first in first out -jono, joka on säieturvallinen [29]. Pipe-funktio luo kaksi rajapinnaltaan identtistä oliota, jotka toimivat prosessien välisen yhteyden kahtena eri päänä. Ensimmäisen olion send-metodia kutsumalla voidaan lähettää viesti, joka voidaan lukea toisen olion recv()-metodilla ja toisin päin. Esimerkissä Ohjelma 6 on toteutettu viestin lähettäminen prosessilta toiselle molemmilla tavoilla. [30]

```
    # -*- coding: utf-8 -*-
2   from multiprocessing import Pipe, Process, Queue

4
    # lisätään taulukko jonoon
6   def jono_funktio(jono):
        jono.put([1, 'kaksi'])
8


10  # lähetetään taulukko yhteyden kautta
    def putki_funktio(conn):
12      conn.send([1, 'kaksi'])
```

```
        conn.close()
14

16  if __name__ == '__main__':
        jono = Queue()
18      jono_prosessi = Process(target=jono_funktio, args=(jono,))
        jono_prosessi.start()
20      # tulostetaan jonon päälimmäinen arvo
        print(jono.get(), 'jono')
22      jono_prosessi.join()

24      # alustetaan yhteyden kaksi päätä omiin muuttujiinsa
        emo_yhteys, lapsi_yhteys = Pipe()
26      putki_prosesi = Process(target=putki_funktio,
                                args=(lapsi_yhteys,))
28      putki_prosesi.start()
        # vastaanotetaan käynnistetyssä prosessissa
30      # lähetetty viesti pääprosessissa
        print(emo_yhteys.recv(), 'putki')
32      putki_prosesi.join()

34  # ohjelma tulostaa:
    # [1, 'kaksi'] jono
36  # [1, 'kaksi'] putki
```

Ohjelma 6. *Prosessien välinen kommunikointi Queuella ja Pipellä [30] mukaillen.*

4.3 Coroutinet ja Taskit

Pythonin asyncio-moduuli toteutta rajapintoja laite- ja verkkokeskeisen koodin samanaikaistamiseen async-await -syntaksia ja taskeja käyttäen [1]. Niillä on mahdollista suorittaa funktioita coroutineina Pythonin omassa event loop -vuorontajassa [31]. Event loop on vapaaehtoinen vuorontaja, eli se suorittaa yhtä asyncio-coroutinea kerrallaan, siirtyen seuraavaan vasta ensimmäisen luopuessa suorituksesta await-avainsanalla. [32] Kuten johdannossa todettiin, tässä työssä ei käsitellä syvemmin event loopin toteutusta sen matalan tason luonteen takia.

Pythonin coroutinet määritellään async-await -syntaksilla. Coroutineksi tarkoitetun funktion määrittelyyn lisätään avainsana ”async” ennen def-avainsanaa ja funktion nimeä. Tällöin funktion sisällä voidaan käyttää await-avainsanaa kutsujen edessä niiden suorituksen päättymisen odottamiseksi ja suoritusvuoron vapaaehtoiseksi luovuttamiseksi seuraavalle coroutinelle, kunnes kutsun suoritus on päättynyt. Kutsun, jonka edessä await-avainsanaa käytetään, tulee olla odotettavissa (engl. awaitable), eli coroutine, Task-olio, tai Future-olio. [33]

Coroutine voidaan suorittaa antamalla se argumentiksi asyncio.run-metodille. Run-metodi luo coroutinelle oman event loopin ja hallitsee funktion sisäisten await-kutsujen vuorontamisesta, mutta ohjelman suoritus säikeessä, jossa run-metodia kutsuttiin ei etene, ennen kuin argumentiksi annetun funktion suoritus päättyy. Jotta useita coroutineita voitaisiin suorittaa vuoronnetusti, tulee niistä luodaa tehtävä eli Task-luokan olio. Tehtävät vastaavat luvun 4.4 future-olioita, sillä ne mahdollistavat coroutineiden suorittamisen seuraamisen ja hallitsemisen, sekä tuloksen odottamisen. [31]

Alla olevissa esimerkeissä havainnollistetaan yksittäistä coroutinea kutsuvan säikeen muun suorituksen keskeytymistä sekä coroutineiden vuorontamista tehtäviä käyttäen. Esimerkissä Ohjelma 7 keskeytyminen näkyy ohjelman lopussa olevasta tulosteesta, jossa ohjelman lopussa oleva ”Loppu” tulostetaan vasta coroutineiden suorituksen päätyttyä. Esimerkissä Ohjelma 8 event loop -vuorontajan toiminta näkyy tulosteen järjestyksessä. Paa_tehtava-coroutine luovuttaa suoritusvuoronsa rivillä 9, jolloin sivu_tehtava-coroutinelle annetaan suoritusvuoro. Esimerkissä käytetään asyncio.gather-funktiota usean coroutinen vuorontamiseksi jokaisen await-kutsumisen sijaan.

```
    # -*- coding: utf-8 -*-
```

```
from asyncio import run


async def tulosta():
    print('Tulosta')


# kutsuu vielä toista coroutinea
async def main():
    print('Main')
    await tulosta()


if __name__ == '__main__':
    print('Alku')
    # keskeytetään ohjelma coroutineiden
    # suorittamiseksi
    run(main())
    print('Loppu')

# ohjelma tulostaa:
# Alku
# Main
# Tulosta
# Loppu
```

Ohjelma 7. *Asyncio.run-metodia suorittavan säikeen suoritusjärjestys*

```
# -*- coding: utf-8 -*-
import asyncio


# coroutine joka luovuttaa
# suoritusvuoron muille keskellä
async def paa_tehtava():
    print('Pää')
    await asyncio.sleep(3)
    print('Pää')


# coroutine joka ei luovuta suoritusta
async def sivu_tehtava():
    print('Sivu')


async def main():
    # käynnistetään coroutineiden suoritus
    # luomalla task molemmista coroutinesta
    await asyncio.gather(
        paa_tehtava(),
        sivu_tehtava()
    )

# käynnistetään event loop
# main-funktiosta
asyncio.run(main())

# ohjelma tulostaa:
# Pää
```

```
    # Sivu
    # Pää
```

Ohjelma 8. *Tehtävien suoritusjärjestyksen vuorontamisen havainnollistaminen*

4.4 Futuret

Concurrent.futures-moduuli on suunnitelu yksinkertaistamaan funktioiden samanaikaista suorittamista säikeiden tai prosessien avulla. [34] Moduulin pääluokat ovat yhteisestä Executor-luokasta periytyvät säikeitä käyttävä ThreadPoolExecutor ja prosesseja käyttävä ProcessPoolExecutor sekä näillä suoritettujen kutsujen tuloksia vastaava Future-luokka. Executor-luokkia alustettaessa niille annetaan argumentteina käytettävissä olevien säikeiden tai prosessien määrä. Alustamisen jälkeen luokkien instanssit voidaan komentaa suorittamaan funktiota submit- tai map-metodeilla. Lisäksi luokat tarjoavat shutdown-metodin resurssien vapauttamiseksi suorituksen päättymisen jälkeen.

Yksittäisen Executor-luokan instanssille annetun submit- tai map-kutsun tulosta vastaava Future-luokka toteuttaa rajapinnan laskennan tilan seuraamiseen cancelled-, done-, ja running-metodeilla, laskennan lopettamiseen cancel-metodilla sekä tuloksen paluuarvon odottamiseen result-metodilla. Luokkaa käyttäen on myös mahdollista liittää tulokseen takaisinkutsufunktio add_done_callback-metodilla, jota kutsutaan tuloksen valmistuttua. [35] Esimerkissä Ohjelma 9 käsitellään Executor-luokkien sekä niiden palauttamien Future-luokan instanssien käyttöä.

```
from concurrent.futures import Executor, Future, \
    ProcessPoolExecutor, ThreadPoolExecutor


# executor-luokilla suoritettava funktio
def palauta_merkkijono(merkkijono):
    return merkkijono


# takaisinkutsufunktio
def tulosta(futuuri):
    print(futuuri.result())


if __name__ == '__main__':
    # käynnistetään säie suorittamaan
    # palauta_merkkijono-funktiota,
    # tulostetaan tuloksen suorituksen
    # tila ja odotetaan vastausta
    with ThreadPoolExecutor(max_workers=1) as executor:
        futuuri = executor.submit(palauta_merkkijono, 'säie')
        print(futuuri.running())
        print(futuuri.result())

    # suoritetaan sama laskenta prosessilla
    # käsitellään tulos erillisessä takaisinkutsufunktiossa
    with ProcessPoolExecutor(max_workers=1) as executor:
        futuuri = executor.submit(palauta_merkkijono, 'prosessi')
        futuuri.add_done_callback(tulosta)

# ohjelma tulostaa:
# False
# säie
```

```
34      # prosessi
```

Ohjelma 9. *Esimerkki funktion kutsumisen samanaikaistamisesta futures-moduulilla*

5. RAJAPINTOJEN KÄYTTÖKOHTEET JA VERTAILU

Työssä tarkasteltavan CPython-tulkin muistinhallinta ei ole säieturvallinen, mikä tarkoittaa, että sillä ei voi suorittaa ohjelmia, jotka käsittelevät sen tuottamaa laitekoodia useassa eri säikeessä. Suojellakseen tulkin tilaa, CPython estää tämän lisäämällä tulkattuun koodiin mutexin eli lukkomuuttujan, jota kutsutaan nimellä Global Interpreter Lock eli GIL. Se lukitaan jonkin säikeen käyttäessä tulkin tuottamaa laitekoodia, sulkien muut säikeet pois, kunnes lukko avataan. Tämä rajoittaan kielen monisäikeisen samanaikaistamisen hyödyntämistä käytännössä eliminoimalla mahdollisuuden käyttää säikeitä CPU-intensiivisissä tehtävissä. [36] Johtuen prosesseja luodessa ja lopettaessa tapahtuvasta erillisen muistialueen käsittelystä ja prosessien välisen kommunikaation suhteellisesta hitaudesta, säikeiden käytön yleiskustannus on kuitenkin pienempi kuin prosessien. [4] Siksi threading-moduulin voi olla nopeampi, kun kohdeongelma on suorittaa useita lyhyitä laskentoja tai suorittaa levy-, laitteisto-, ja verkkokeskeisiä operaatioita, joissa suorittaja odottaa ulkoista resurssia.

CPU-intensiivisissä tehtävissä prosessipohjainen samanaikaisuus, kuten multiprocessing-moduuli tai concurrency.futures-moduulin ProcessPoolExecutor-luokka on nopeampi kuin vastaava monisäikeisyyteen pohjaava toteutus. [37] Verrattuna kuitenkin luvussa 4.4 käsiteltyyn concurrent.futures-moduulin ProcessPoolExecutor-luokkaan, multiprocessing-moduulin tarjoamien Process- ja Pool-luokkien rajapinnat vaativat käyttäjältä enemmän kirjoittamista ja tarjoavat vähemmän työkaluja suorituksen tilan seurantaan ja hallintaan. Lisäksi prosessipohjainen ja monisäikeinen samanaikaisuus vaativat ohjelmoijaa toteuttamaan itse moduulien tarjoamia rajapintoja käyttäen prosessien tai säikeiden välisen kommunikaation ja luvun 2.5 ongelmatapausten ratkaisun. [28][30]

Luvussa 4.3 käsitelty asyncio-moduuli mahdollistaa ohjelman suorittamisen samanaikaistamisen multiprocessing- ja threading-moduuleja helpommalla rajapinnalla ilman prosessien ja säikeiden vaatimia turvamekanismeja tai kommunikaatiota. Asyncio-moduulin event loop -vuorontajan käyttö kuitenkin vaatii, että minkään coroutinen suoritus ei kestä liian kauaa, koska event loop käyttää vain yhtä säiettä. Tämä hidastaisi käytännössä muiden coroutineiden suoritusta. Tässä tapauksessa asyncio tarjoaa rajapinnan säikeiden ja concurrent.futures-moduulin executor-luokkien käyttöön. [38]

Luvussa 4.4 käsitellyt Concurrent.futures-moduulin executor-luokat on suunniteltu toteuttamaan yksinkertaisempi asynkroninen rajapinta Python-koodin samanaikaistamiseen säikeillä ja prosesseilla sekä tulosten ja samanaikaisen suorituksen tilan hallintaan. [8] Rajapinta ei kuitenkaan ota kantaa säikeiden tai prosessien väliseen kommunikointiin, tai luvussa 2.5 käsiteltyihin samanaikaistamisen ongelmatapauksiin ja niitä ratkaiseviin mekanismeihin. [35]

6. YHTEENVETO

Työssä käsiteltiin Pythonin standardikirjaston samanaikaistamiseen, sekä samanaikaistamisen helpottamiseen tarkoitettuja asynkronisia rajapintoja kertomalla niiden rakenteesta ja toiminnasta. Rajapintojen käyttöä selvennettiin esittämällä niiden taustalla olevia käyttöjärjestelmän mekanismeja sekä osoittamalla mahdollisia ongelmakohtia samanaikaisuuden toteuttamisessa näitä mekanismeja käyttäen. Rajapinnoista esitettiin koodiesimerkkejä synkronisen koodin ja vastaavan samanaikaistetun toteutuksen erojen havainnollistamiseksi, sekä samanaikaisten ja asynkronisten rajapintojen käytön ja erojen hahmottamisen helpottamiseksi.

Työssä pohdittiin myös esiteltyjen rajapintojen ja niitä käyttävien mekanismien hyviä ja huonoja puolia nopeuden ja rajapinnan käytön helppouden näkökulmasta. CPU-keskeisen laskennan samanaikaistamiseen tehokkain on prosesseihin pohjaava multiprocessing-moduuli. Kevyempiin laskentatehtäviin, latenssin pienentämiseen ja laitteisto- ja verkkokeskeiseen laskentaan tehokkain on threading-moduuli. Asyncio-moduuli on helppokäyttöisin käyttökohteisiin, joissa koko ohjelma voidaan rakentaa asynkroniseksi. Säikeiden ja prosessien käyttöön vähiten kirjoitettavaa koodia vaativa ja eniten työkaluja tarjoava rajapinta on concurrent.futures-moduuli.

Python on jatkuvasti kehittyvä kieli, jonka kehitystä ajaa laaja yhteisö. Tämän takia osa tarkastelluista rajapinnoista ei ole täysin vertailukelpoisia, koska niiden kehityksessä on tähdätty eri tarpeiden täyttämiseen. Lisäksi Pythonin standardikirjastoon toteutetut rajapinnat nojaavat usein jo olemassa oleviin kirjastoihin ja niissä toteutettuihin ratkaisuihin, jolloin standardikirjaston toteutus saattaa olla jopa huonommin optimoitu tai suunniteltu. Toisaalta kopioinnin taustalla on osa Pythonin suunnitteluperiaatetta tuoda tärkeitä ja usein käytettyjä työkaluja standardikirjastoon.

Työ ei käsitellyt threading-moduulissa olleita apuluokkia ja funktioita, jotka on toteutettu vastaamaan osaa luvussa 2.5 käsiteltyjä samanaikaisuuden ongelmatapauksia ja niiden ratkaisuja. Näillä saattaa olla merkitystä koodin suorituskykyyn sekä kirjoittamisen helppouteen. Työ ei myöskään käsitellyt koko Pythonin syntaksia, vaan ainoastaan työn esimerkkeihin vaadittavia kohtia. Lisäksi luvussa 5 esitettyjä väitteitä rajapintojen käytettävyydestä olisi voinut pohjustaa ja tutkia paremmin.

LÄHTEET

[1] Python Software Foundation, asyncio — Asynchronous I/O, 2019. Saatavissa: https://docs.python.org/3/library/asyncio.html. Viitattu 3.6.2019.

[2] Oracle Corporation, Multithreaded Programming Guide, 2010. Saatavissa: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html. Viitattu 3.6.2019.

[3] V. Sharma, Learning Scala Programming, 1. painos, Packt Publishing, 2018. Saatavissa: https://learning.oreilly.com/library/view/learning-scala-programming/9781788392822/3b58278f-a0b6-4a32-8e14-5212f1ebe127.xhtml. Viitattu 3.6.2019.

[4] I. Haikala, H-M. Järvinen, Käyttöjärjestelmät, 2. painos, Talentum, 2004.

[5] D. E. Culler, J. P. Singh, Parallel Computer Architecture, Morgan Kaufman Publishers, Inc., 1999, 28–36 p.

[6] J. Bartel, Non-preemptive Multitasking, The Computer Journal, 1988, 37 p. Saatavissa: https://ia802905.us.archive.org/13/items/the-computer-journal-

[7] M. Conway, Design of a separable transition-diagram compiler, Communications of the ACM, 1963, 1–3 p. Saatavissa: https://dl.acm.org/citation.cfm?id=366704

[8] B. Quinlan, PEP 3148 -- futures - execute computations asynchronously, 16.10.2009. Saatavissa: https://www.python.org/dev/peps/pep-3148/. Viitattu 3.6.2019.

[9] P. Haller, A. Prokopec et al., SIP-14 - Futures and Promises, 2017. Saatavissa: https://docs.scala-lang.org/sips/completed/futures-promises.html. Viitattu 3.6.2019.

[10] Python Software Foundation, Barrier Objects, 2019. Saatavissa: https://docs.python.org/3/library/threading.html#barrier-objects. Viitattu 3.6.2019.

[11] Python Software Foundation, Why was Python created in the first place? 2019. Saatavissa: https://docs.python.org/3/faq/general.html#why-was-python-created-in-the-first-place. Viitattu 3.6.2019.

[12] A.M. Kuchling, PEP 206 -- Python Advanced Library, 2019. Saatavissa: https://www.python.org/dev/peps/pep-0206/#batteries-included-philosophy. Viitattu 3.6.2019.

[13] D. Kuhlman, "A Python Book: Beginning Python, Advanced Python, and Python Exercises", 1.1.2 ch. Saatavissa: https://www.davekuhlman.org/python_book_01.pdf. Viitattu 3.6.2019.

[14] M. Lutz, Programming Python, 4. painos, O'Reilly Media, Inc., 2010, Preface kpl. Saatavissa: https://learning.oreilly.com/library/view/programming-python-4th/9781449398712/pr02.html. Viitattu 3.6.2019.

[15] Python Software Foundation, Glossary, 2019. Saatavissa: https://docs.python.org/3.7/glossary.html#term-bytecode. Viitattu 3.6.2019.

[16] P. Guo, CPython internals: A ten-hour codewalk through the Python interpreter source code, 2014. Saatavissa: http://www.pgbovine.net/cpython-internals.htm. Viitattu 3.6.2019.

[17] O. Ike-Nwosu, Inside The Python Virtual Machine, Leanpub, 2019, 1 kpl. Saatavissa: https://leanpub.com/insidethepythonvirtualmachine/read. Viitattu 3.6.2019.

[18] Python Software Foundation, Using the Python Interpreter, 2019. Saatavissa: https://docs.python.org/3/tutorial/interpreter.html. Viitattu 3.6.2019.

[19] A. Holkner, J. Harland, Evaluating the dynamic behaviour of Python applications, ACSC '09 Proceedings of the Thirty-Second Australasian Conference on Computer Science, 2009, vol. 91, 19 s. Saatavissa: https://dl.acm.org/citation.cfm?id=1862665. Viitattu 3.6.2019.

[20] Python Software Foundation, Command line and environment, 2019. Saatavissa: https://docs.python.org/3/tutorial/modules.html. Viitattu 3.6.2019.

[21] Python Software Foundation, Modules, 2019. Saatavissa: https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH. Viitattu 3.6.2019.

[22] G. van Rossum, PEP 8 -- Style Guide for Python Code, 2009. Saatavissa: https://www.python.org/dev/peps/pep-0008/#indentation. Viitattu 3.6.2019.

[23] Python Software Foundation, Execution model, 2019. Saatavissa: https://docs.python.org/3/reference/executionmodel.html. Viitattu 3.6.2019.

[24] Python Software Foundation, Concurrent Execution, 2019. Saatavissa: https://docs.python.org/3/library/concurrency.html. Viitattu 3.6.2019.

[25] Microsoft, Asynchronous Programming with Async and Await (C# and Visual Basic), 2017. Saatavissa: https://msdn.microsoft.com/en-us/library/hh191443.aspx. Viitattu 3.6.2019.

[26] HHVM Community, Asynchronous Operations: Introduction, 2019. Saatavissa: http://docs.hhvm.com/manual/en/hack.async.php. Viitattu 3.6.2019.

[27] P. Haller, J. Zaugg, SIP-22 – Async, 2016. Saatavissa: http://docs.scala-lang.org/sips/pending/async.html. Viitattu 3.6.2019.

[28] Python Software Foundation, threading — Thread-based parallelism, 2019. Saatavissa: https://docs.python.org/3/library/threading.html. Viitattu 3.6.2019.

[29] Python Software Foundation, queue — A synchronized queue class, 2019. Saatavissa: https://docs.python.org/3/library/queue.html#queue.Queue. Viitattu 3.6.2019.

[30] Python Software Foundation, multiprocessing — Process-based parallelism, 2019. Saatavissa: https://docs.python.org/3.7/library/multiprocessing.html. Viitattu 3.6.2019.

[31] Python Software Foundation, Coroutines and Tasks, 2019. Saatavissa: https://docs.python.org/3/library/asyncio-task.html. Viitattu 3.6.2019.

[32] Python Software Foundation, Event Loop, 2019. Saatavissa: https://docs.python.org/3/library/asyncio-eventloop.html. Viitattu 3.6.2019.

[33] Y. Selivanov, PEP 492 -- Coroutines with async and await syntax, 9.5.2015. Saatavissa: https://www.python.org/dev/peps/pep-0492/. Viitattu 3.6.2019.

[34] B. Quinlan, PEP 3148 -- futures - execute computations asynchronously, 16.10.2009. Saatavissa: https://www.python.org/dev/peps/pep-3148/#motivation. Viitattu 3.6.2019.

[35] Python Software Foundation, concurrent.futures — Launching parallel tasks, 2019. Saatavissa: https://docs.python.org/3/library/concurrent.futures.html. Viitattu 3.6.2019.

[36] D. Beazley, Understanding the Python GIL, 20.2.2019. Saatavissa: http://www.dabeaz.com/GIL/. Viitattu 3.6.2019.

[37] J. Noller, R. Oudkerk, PEP 371 -- Addition of the multiprocessing package to the standard library, 6.5.2019. Saatavissa: https://www.python.org/dev/peps/pep-0371/. Viitattu 3.6.2019.

[38] Python Software Foundation, Developing with asyncio, 2019. Saatavissa: https://docs.python.org/3/library/asyncio-dev.html. Viitattu 3.6.2019.